STA 32 R Handout 4, Data Types in R

1 Basic Data Types

R has a wide variety of data types including scalars, vectors (numerical, character, logical), matrices, data frames, and lists.

1.1 Vectors

Using our old example:

```
X = c(67,63,65,71,72,68,60,59,67,66,75,74,69)
Y = c("F","F","F","M","M","M","F","F","M","F","M","M","F")
```

We have created two vectors of different type. To check what type of data our variable is, we can use the `class` function.

```
class(X)
[1] "numeric"
class(Y)
[1] "character"
```

If we put characters and numbers into the same vector, R will treat the vector as a character vector.

```
Z = c("S","T","A",32)
class(Z)
[1] "character"
```

In R, `TRUE` and `FALSE` are logical values. We do not need to add quotation marks around them when we use them.

```
logic1 = c("TRUE", "FALSE")
logic2 = c(TRUE, FALSE)
class(logic1)
[1] "character"
class(logic2)
[1] "logical"
```

To check specifically whether a given vector is numerical, character, or logical, we can use `is.numeric`, `is.character`, and `is.logical` respectively.

```
is.numeric(X)
TRUE
is.character(X)
FALSE
```

1.2 Matrix

There are a few ways to create a matrix.

- Using `rbind` or `cbind`.
  The `rbind` function combines vectors by rows (row bind), where `cbind` combines vectors by columns (column bind). To use these two commands, you need to have vectors of the same length.

  ```
  A = c(1,2,3); B = c(4,5,6)
  rbind(A,B)
    [,1] [,2] [,3]
  A    1    2    3
  B    4    5    6
  cbind(A,B)
       A B
  [1,] 1 4
  [2,] 2 5
  [3,] 3 6
  ```

- Using the `matrix` function.

  ```
  D = c(A,B)
  [1] 1 2 3 4 5 6
  matrix(D)
  matrix(D,nrow = 2)
       [,1] [,2] [,3]
  [1,]    1    3    5
  [2,]    2    4    6
  matrix(D,ncol = 3)
  matrix(D,nrow = 2, ncol = 3)
  matrix(D,nrow = 2, byrow = TRUE)
       [,1] [,2] [,3]
  [1,]    1    2    3
  [2,]    4    5    6
  ```

  The first command simply combines two numeric vector together. The second command will take the elements in `D` and use them as the elements in the matrix. The third, fourth and fifth commands produce the same result: a matrix of dimension 2x3. Note `R` will fill up the matrix column by column. If we want to fill in the matrix row by row, we add the argument **byrow = TRUE**. If you want, you can give names to the rows and columns.

```
rnames = c("R1", "R2"); cnames = c("C1", "C2", "C3")
matrix(D, nrow = 2, dimnames = list(rnames, cnames))
   C1 C2 C3
R1  1  3  5
R2  2  4  6
```

Of course, the dimension of these name vectors need to match with the dimension of the matrix. Just like above, we can check whether a given variable is a matrix or not by using the `is.matrix` function.

1.3 Data Frames

We have seen this already when we combine two vectors using the `data.frame` function. To check explicitly whether a variable is a data frame or not, we use `is.data.frame`. Note if we use `cbind` or `rbind` to combine two vectors of different types, we don't get a data frame; we get a matrix instead.

```
Dm = data.frame(X,Y)
class(Dm)
T = rbind(X,Y)
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
X "67" "63" "65" "71" "72" "68" "60" "59" "67" "66"  "75"  "74"  "69"
Y "F"  "F"  "F"  "M"  "M"  "M"  "F"  "F"  "M"  "F"   "M"   "M"   "F"
class(T)
[1] "matrix"
```

Of course, you can't really do any matrix arithmetic on this variable.

1.4 List

This is probably one of the most powerful data type in `R`, but also one of the hardest one to use/master. A list is an ordered collection of objects (components), it allows you to gather a variety of (possibly unrelated) objects under one name. For example, you can put a numeric vector, a character vector, a matrix, and a data frame, all into one object using list.

```
L = list(X, Y, Dm, T)
[[1]]
 [1] 67 63 65 71 72 68 60 59 67 66 75 74 69

[[2]]
 [1] "F" "F" "F" "M" "M" "M" "F" "F" "M" "F" "M" "M" "F"

[[3]]
   X Y
1 67 F
2 63 F
```

```
3  65 F
4  71 M
5  72 M
6  68 M
7  60 F
8  59 F
9  67 M
10 66 F
11 75 M
12 74 M
13 69 F

[[4]]
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
X "67" "63" "65" "71" "72" "68" "60" "59" "67" "66"  "75"  "74"  "69"
Y "F"  "F"  "F"  "M"  "M"  "M"  "F"  "F"  "M"  "F"   "M"   "M"   "F"
class(L)
[1] "list"
class(L[[1]])
[1] "numeric"
```

To access the elements in the list, we need to use the [[ ]]. For example, to grab the character vector that is in the second position of the list, we do `L[[2]]`, and to also grab the fourth element in that character vector (which is the "M"), we do `L[[2]][4]`. Notice we only use a single set of [ ] in the second part, because `L[[2]]` is no longer a list. We can give names to each item in the list by doing the following

```
L = list(numeric = X, character = Y, dataset = Dm, matrix = T)
$numeric
 [1] 67 63 65 71 72 68 60 59 67 66 75 74 69

$character
 [1] "F" "F" "F" "M" "M" "M" "F" "F" "M" "F" "M" "M" "F"

$dataset
    X Y
1  67 F
2  63 F
3  65 F
4  71 M
5  72 M
6  68 M
7  60 F
8  59 F
9  67 M
10 66 F
```

```
11 75 M
12 74 M
13 69 F

$matrix
  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
X "67" "63" "65" "71" "72" "68" "60" "59" "67" "66"  "75"  "74"  "69"
Y "F"  "F"  "F"  "M"  "M"  "M"  "F"  "F"  "M"  "F"   "M"   "M"   "F"
```

Note we don't need to add in "" for the names. To access the character vector, we can do `L$character` (just like how we access a particular column in a data frame). Then to grab the fourth element in that character vector, we simply do `L$character[4]`.

1.5 Others

There are a few other types that we use, such as `factor`, `array`, `table`. You are welcomed to explore them using the help pages.

2 Converting Between Different Data Types

Sometimes we want to convert a data type to a different data type. For example, we have a numeric vector and we want to convert it into a matrix so we can apply matrix operations on it. We have several commends to do them: `as.numeric, as.character, as.matrix, as.data.frame, as.list, as.factor`, etc.

```
X
[1] 67 63 65 71 72 68 60 59 67 66 75 74 69
charX = as.character(X)
charX
[1] "67" "63" "65" "71" "72" "68" "60" "59" "67" "66" "75" "74" "69"
class(charX)
[1] "character"
as.matrix(X)
      [,1]
 [1,]   67
 [2,]   63
 [3,]   65
 [4,]   71
 [5,]   72
 [6,]   68
 [7,]   60
 [8,]   59
 [9,]   67
[10,]   66
[11,]   75
```

```
[12,]   74
[13,]   69
```

By default, R will convert a vector into a column matrix if you use as.matrix. R tends to do everything in terms of columns, so it's best to check the result is in the format you want.

3 sapply, lapply and apply

3.1 sapply

We have seen the use of sapply when we don't want to use a for loop. However, sapply can do more than that. Another great use of sapply is that it can loop over each element in the supplied object and apply the given operations to them. We can see a basic example in the last handout, let's see another one.

```
sapply(L, function(i) class(i))
     numeric    character      dataset       matrix
   "numeric"  "character" "data.frame"     "matrix"

numlist = list(c(5,2,6,7),c(1,4,2,1,1,9),rep(0.5,3))
sapply(numlist, function(i) length(i))
[1] 4 6 3
```

In the first example, sapply loops over each element in the list L and checks the class for that element. Note we use class(i). What sapply is doing is it treats each element in the list as an object named i, and use the class function like how we did above. In the second example, numlist is a list that contains three numeric vector, and in the sapply function I checks the length of each vector.

3.2 lapply

lapply is essentially the same as sapply, except sapply will try to suppress the output to the simplest format, while lapply always returns a **list**.

```
s.example = sapply(numlist, function(i) length(i))
class(s.example)
[1] "integer"
s.example
[1] 4 6 3
l.example = lapply(numlist, function(i) length(i))
class(l.example)
[1] "list"
l.example
[[1]]
[1] 4
```

```
[[2]]
[1] 6

[[3]]
[1] 3
```

Note that if the resulting vector is all integers, we have a different data type called **integer**.

3.3 apply

`apply` is used when you want to apply some functions to all a particular dimension in an array. For example, you want to find the mean of each row in a matrix, or you want to add up all the elements in each column in a matrix. Of course, you can use the `rowMeans` and `colSums` function respectively, but let's still see how to use this `apply` function.

```
DD = matrix(D, nrow = 2, byrow = TRUE)
apply(DD, 1, mean)
[1] 2 5
apply(DD, 2, sum)
[1] 5 7 9
```

The **1** in the second line represents the **row** dimension, where the **2** represents the **column** dimension. The **mean** in the `apply` function is the build in `R` function `mean`, and the same goes to the **sum** in the fourth line. You can write your own function when you do your own work. For functions with extra arguments, we can use the following example as a guide:

```
x = cbind(x1 = 3, x2 = c(4:1, 2:5))
cave = function(x, c1, c2) c(mean(x[c1]), mean(x[c2]))
apply(x, 1, cave,  c1 = "x1", c2 = c("x1","x2"))
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]  3.0    3  3.0    3  3.0    3  3.0    3
[2,]  3.5    3  2.5    2  2.5    3  3.5    4
```

(This example is taken from the help page) The cave function takes in a numeric vector (x), and two names (c1, c2). It then computes the means x[c1] and x[c2], and outputs them as a numeric vector. c1 and c2 can be a character name, or can be indices of elements. In the `apply` function, we input the matrix **x**, loop over each row (because we specify **1**), and apply the function `cave`. For each row, we calculate the mean of the "x1 column" and the mean of the "x1 x2 columns". There are two things to notice here:

1. The `apply` function (same as `sapply`) will stack the outputs by columns if we have multiple returns, so each column is a new set of solutions.

2. In the `cave` function, the first argument is set to be **x**. Whenever we do something to a function (such as `apply` here, or optimization if you ever use it) and an input is related to a different function, you set this argument to be the first argument in the parent function (in this case, the parent function is the `cave` function).